SWDE - Software Development
Zusammenfassung FS 2019

Maurin D. Thalmann

25. Februar 2019

Inhaltsverzeichnis

1 Buildautomatisation

1.1 Sie kennen die Vorteile eines automatisierten Buildprozesses

- Automatisierter Ablauf, keine Interaktion mehr benötigt
- Reproduzierbare Ergebnisse
- lange Builds können auch über Nacht laufen
- Unabhängig von Entwicklungsumgebung

1.2 Sie können verschiedene Beispiele von Buildwerkzeugen benennen

Make (für C/C++ Projekte), Urvater der Build Tools,
hohe Flexibilität, gewöhnungsbedürftige Syntax

Ant Java mit XML

Maven Java mit XML

Buildr Ruby-Script

Gradle Groovy Script mit DSL

Bazel Java mit Python-like Scripts

1.3 Sie beherrschen die Anwendung eines ausgewählten Buildwerkzeuges (Apache Maven)

Beherrschen muss man es selber, es kann entweder aus der Shell (Terminal/Konsole) verwendet werden oder aus den integrierten Funktionen in der IDE selbst.

1.4 Sie sind mit den wesentlichen Konzepten von Apache Maven vertraut

Deklaration des Projektes in XML, zentrales Element pro Projekt ist das **Project Object Model (POM)**, welches Metainformationen, Plugins und Dependencies definiert. Basiert auf einem globalen, binären Repository. Plugins werden durch Dependencies dynamisch ins lokale Repository geladen ($HOME/.m2/repository)
Bei einem Buildprozess durchläuft ein Projekt einen Lifecycle mit folgenden Phasen:

validate validiert Projektdefinition

compile Kompiliert die Quellen

test Ausführen der Unit-Tests

package Packen der Distribution

verify Ausführen der Integrations-Tests

install Deployment im lokalen Repository

deploy Deployment im zentralen Repository

2 Modularisierung - Module, Komponenten, Schnittstellen

2.1 Sie wissen, was unter dem Begriff Modularisierung zu verstehen ist

«Ein grosses Ganzes in mehrere, sich abgeschlossene Einheiten (Module) aufteilen»
Flexible Zusammenstellung, Durchführung und Prüfung der einzelnen Module. Zwischen den Modulen können aber auch Abhängigkeiten bestehen.

2.2 Sie kennen die Begriffe Modul, Library, Komponenten und Schnittstelle auf der Ebene des Softwaredesigns

Kleinste Einheit: Klasse (Methoden/Daten/Attribute) $\leftrightarrow$ **Grösste Einheit:** vollständiges Softwaresystem

Modul In sich abgeschlossene und austauschbare Einheiten,
soll nur über seine Schnittstellen verwendet werden können $\rightarrow$ lose Kopplung;
Starke Kohäsion (möglichst in sich abgeschlossene Aufgabe erfüllen) $\rightarrow$ Information Hiding

Schnittstelle lässt Module untereinander interagieren / austauschen

Komponente strengere Form eines Moduls

Library Eine Sammlung thematisch zusammengehörender Funktionen (z.Bsp. Kalendermodul, Trigonometriemodul, etc.)

Die einzelnen Begriffe werden in einem späteren Lernziel ausführlicher beschrieben.

2.3 Sie können die Begriffe auf verschiedenen Abstraktionsebenen in einen sinnvollen Zusammenhang und Kontext setzen

Kopplung Ausmass der Kommunikation zwischen Modulen (Abhängigkeit zw. Modulen)

Kohäsion Ausmass der Kommunikation innerhalb eines Moduls (interner Zusammenhalt)

Ziel $\rightarrow$ Maximierung der Kohäsion, Minimierung der Kopplung

Gruppierung Modulen mit gemeinsamen Eigenschaften als Gruppe handhaben.
Beispiel: Modul für Datenexport in versch. Formate

Hierarchie (Rekursiv) Modul fasst mehrere (Sub-) Module zu einem zusammen.
Beispiel: Persistenzmodul als Datenspeicher mehrerer Entitäten

Geschichtet Modul(-gruppen) können logische Kette bilden, die vertikal als Schichten abgebildet werden.
Beispiel: Schichtenarchitektur, OSI-Referenzmodell, etc.

2.4 Sie sind in der Lage ein System zu analysieren und darin sinnvolle Module zu identifizieren

2.4.1 Modul

Kriterien für Entwurf von Modulen:
Zerlegbarkeit / Dekomposition
möglichst unabhängig voneinander, können einzeln genutzt/wiederverwendet werden
Kombinierbarkeit
sollen in anderem Umfeld wieder einsetzbar sein
(Zerlegbar, um auf andere Art wieder kombiniert werden zu können)
Verständlichkeit
unabhängig und in sich abgeschlossen verständlich sein, kann aber trotzdem hohe Komplexität erreichen.
Stetigkeit / Stabilität / Kontinuität
Struktur soll sich nicht stetig verändern, Aufteilung soll robust gegenüber Änderungen sein. Änderungen sollen sich auf eine minimale Anzahl Module beschränken.

Arten von Modulen:
Bibliothek/Library: beschrieben unter 2.2
Abstrakte (komplexe) Datentypen: Modul implementiert neuen komplexen Datentyp und stellt darauf definierte Operationen zur Verfügung (Bsp. Komplexe Zahlen, Koordinatendarstellung, etc.)
Modellierung/Abstraktion physischer Modelle: Modul abstrahiert reales, physisch existierendes System (z.Bsp. Sensor, Gerätetreiber, Anzeigemodul, etc.)
Modellierung/Abstraktion logisch-konzeptioneller Systeme: Modul abstrahiert ein nur rein logisch existierendes System und macht es für eine höhere Abstraktionsebene nutzbar (z.Bsp. Grafikmodule, Datenbankmodule, Messaging, GUI-Module, etc.)

Definition eines Moduls:
Verhalten: Funktionalität des Moduls?
Export: Was bieten wir, Schnittstelle, um das Verhalten des Moduls für andere Module verfügbar zu machen.
Import: Was brauchen wir, von welchen Schnittstellen ist das Modul evtl. abhängig? (Dependencies)

Herausforderung bei Modulen:
Basiskonzepte: Hohe Kohäsion, lose Kopplung, starke Datenkapselung & Information Hiding
Vier Kriterien: Zerlegbarkeit, Kombinierbarkeit, Verständlichkeit & Stetigkeit
Verschiedene Arten: Bibliotheken, abstrakte Datentypen, physische / logische Modelle, Komponenten, etc.

2.4.2 Komponente

> *Eine Softwarekomponente ist ein Softwareelement, das zu einem bestimmten **Komponentenmodell** passt und entsprechend einem Composition Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann.*

Eine **Komponente** erfüllt strengere Kriterien als ein Modul und benötigt meist einen **Kontext**. Komponenten bedienen sich spezifischer Laufzeitumgebungen (Bsp. Container) in welche die Komponenten integriert (installiert, deployed, etc.) werden und dort lauffähig sind. (Container stellen den Komponenten Basisdienste z.Bsp. für Lifecycle und Kommunikation bereit)
Komponenten bedienen sich spezifischer Laufzeitumgebungen (z.Bsp. Container) in welche die Komponenten integriert (installiert, deployed, etc.) werden und dort lauffähig sind (Container stellen den Komponenten Basisdienste z.Bsp. für Lifecycle und Kommunikation bereit)
Komponenten können teilweise **dynamisch zur Laufzeit** ergänzt/entfernt/ausgetauscht werden
→ «Hot-Deployment» / Plugin-Mechanismen
Komponenten bieten Funktionalität an und sind von Funktionalität des eingesetzten Komponentenmodells (Framework/Produkt) **abhängig**.

Komponentenmodelle: konkrete Ausprägungen des Paradigmas der komponentenbasierten Entwicklung in Form eines Standards, Frameworks, Produktes. Schnittstellen für Interaktion und Komposition von Komponenten festlegen: wie kommunizieren Komponenten untereinander/mit dem Container Idealerweise: Komponentenmodell unabhängig von Gremium standardisiert (kann somit in unterschiedlichen Ausprägungen von versch. Herstellern implementiert/genutzt werden)

> **Beispiele**: Microsoft DCOM/ActiveX/.NET Remoting Services (WCF), CORBA (Common Object Request Broker Architecture), Enterprise Java Beans, OSGi (Open Services Gateway initiative / Alliance)

Komponente wird wie Modul über *Verhalten, Export, Import* definiert.
Zusätzlich wird ein **Kontext** verlangt: definiert notwendige Rahmenbedingungen, die für Betrieb der Komponente notwendig sind.

2.4.3 Schnittstelle

Schnittstellen werden konsequent für die kontrollierte Kommunikation zwischen Modulen oder Komponenten verwendet. **Vorteile** davon:

- Schnittstelle ist einfach verständlich, einfacher als die Implementierung.
- Schnittstellen helfen Abhängigkeiten zu reduzieren, vermeiden Abhängigkeiten zur Implementierung.
- Schnittstellen erleichtern Wiederverwendung.

Beziehungen zwischen einzelnen Teilen einer Software werden über Schnittstellen realisiert. Module konzentrieren sich auf ihre lokalen Probleme, Architektur definiert und hält Fäden (Beziehungen) des Systems zusammen.
Schnittstellen sollen minimal und schmal sein → aussagekräftige Methoden, präzis typisierte Parameter, **Methoden** sollen möglichst:

- keine Überschneidungen haben
- keine globalen Daten verwenden
- statuslos (stateless) sein

Service: abstrahierte Schnittstelle, definiert sich primär über Fachlichkeit, dahinterliegende Technik idealerweise vollständig isoliert (Bsp. Webservice, wird über Web-Protokolle angeboten und abstrahiert die Implementation [Plattform, Sprache, Technologie] vollständig)
API: *(Application Programming Interface)*, technisch orientierte Schnittstelle, welche die Anbindung einer Komponente auf Quellcodeebene definiert (Bsp. JDBC [Java Database Connectivity], einheitliche Schnittstelle zur Kommunikation mit versch. DBMS)
Ebene **Objektorientiertes Design**: Schnittstelle = Java Interface
Ebene **Modularisierung**: Schnittstelle = logische Zusammenfassung versch. Artefakte (Klassen, Interfaces, Konfigurationsdateien, Doku etc.)

2.5 Sie kennen verschiedene organisatorische und technische Varianten um eine sinnvolle Modularisierung in der Entwicklung und im Deployment einzusetzen

2.5.1 Java 8

Module/Komponenten mit Klassen und Interfaces realisiert, Deployment meist als JAR. Klassen können sich an «Java Bean Spezifikation» halten
→ Default-Konstruktor, Setter/Getter, PropertyChange, Serialisierbar, etc.
Schnittstellen mit Java-Interfaces (zu class-Dateien kompiliert)
Komplexere Schnittstellen: mehrere Interfaces in Package zusammenfassen
Java 1.8 unterstützt selber keine Modularisierung
Information Hiding (einzelne Elemente vor Zugriff schützen) durch Packages, Sichtbarkeiten und Import, ermöglicht Zusammenfassen von Klassen/Interfaces in Gruppen, aber keine explizite Möglichkeit, Exports & Dependencies zu deklarieren
manifest.mf enthält Infos zu Identifikation, Herkunft und Version
Schnittstellen in getrennten JAR's (Modulen) verteilen → einfacher Austausch unterschiedlicher Implementationen
Workaround: Namenskonventionen und hohe Disziplin

2.5.2 Java 9

Modularisierung möglich, drei Ziele im Vordergrund:

- **Reliable Configuration:** fehleranfälligen Classpath durch auf Modul-Abhängigkeiten basierenden Modul-Path ablösen
- **Strong Encapsulation:** Modul definiert explizit sein öffentliches API. Auf alle restlichen Klassen ist von aussen kein Zugriff mehr möglich (auch wenn public).
- **Scalable Platform:** Java-Plattform selber wurde modularisiert, so können für Anwendungen individuell angepasste, schlankere Runtime-Images gebaut werden.

Weiteres zu Modularisierung in Java 9:

- Java-Packages neu in Modulen zusammenfassbar (zusätzliche Strukturebene in der Dateiablage, eindeutige Namensgebung analog zu Packages)
- Pro Modul wird ein *module-info.java* definiert (explizite Definition von Imports/Exports/Abhängigkeiten)
- Start einer Applikation: Laufzeitprüfung wird ausgeführt, ob alle notwendigen Komponenten vorhanden sind.
- Ende der «JAR-Hell»: Neues Format *jmod*, Class-Path wird durch Modul-Path abgelöst
- Vollständig rückwärtskompatibel

3 Versionskontrollsysteme - Source Code Management (SCM) / Version Control Systems (VCS)

3.1 Sie kennen die Aufgaben eines Versionskontrollsystems und können grundlegend damit arbeiten

Grundlegende Arbeit:

checkout lokale Arbeitskopie eines Projekts erstellen

update Änderungen Dritter in Arbeitskopie aktualisieren

log Bearbeitungsgeschichte eines Artefakts ansehen

diff verschiedene Revisionen miteinander vergleichen

commit / **checkin** Artefakte ins Repository schreiben → aussagekräftiger Kommentar!

Tagging: Revisionsstand mit Namen markieren, Marke oder Version: 1.5.2beta o.ä. Nützlich bei Release eines Produkts (aber auch meilensteine, Testversionen, Auslieferungszustände, etc.) wird unterschiedlich realisiert.

Branching: Parallele, voneinander getrennte Entwicklungszweige (für Bugfixing, Prototypen, Tests, Experimente, nachvollziehbare Änderungsworkflows, etc.) Bei Nicht-Wegwerf-Entwicklungen später Merging möglich/notwendig.

Ausschliesslich Quell-Artefakte werden verwaltet, **NIE** generierte/erzeugt Artefakte einchecken, können mit Hilfe der SCM ignoriert werden (.gitignore)

3.2 Sie kennen die verschiedenen Konzepte und Arten von Versionskontrollsystemen

- Zentrale oder verteilte Systeme
- Optimistische und pessimistische Lockverfahren
- Versionierung auf Basis einer Datei, Verzeichnisstruktur oder der Änderung (changeset)
- Transaktionsunterstützung (vorhanden oder nicht)
- Verschiedene Zugriffsprotokolle und Sicherheitsmechanismen
- Integration in Webserver (vorhanden oder nicht)

3.3 Sie können mit verschiedenen (Client-)Werkzeugen von Versionskontrollsystemen alleine und im Team arbeiten

CVS UT-Versionskontrollsystem, stabil, wenig Fehler, einfache Anwendung, ABER nur dateibasierend, Verzeichnisstruktur nicht versioniert, unterscheidet zwischen Text- und Binärdateien, Ablage von Binärdateien platzintensiv, keine Transaktionen

Subversion Transaktionsorieniert, versioniert ganze Verzeichnisstruktur, optimierte/effiziente Speicherung und Übertragung, Repositorystruktur frei wählbar (für Experten flexibler, für Anfänger schwieriger), Integration in Webserver möglich, aber Branching und Tagging technisch eig. Kopien/Links

git verteiltes System, primär lokale Arbeit, beliebig viele Server/Repos möglich, auch rein lokal einsetzbar, skaliert, Integration mit zusätzlichen Web-Applikationen, erfordert aber ein solides Konzept, für Einsteiger schwierig, da sehr mächtig und viele Funktionen